Laboratorul 4

- Responsabil laborator: Mihai Nan
- Profesor coordonator: Carmen Odubasteanu

Principii POO

Intre obiectele lumii care ne inconjoara exista, de multe ori, anumite relatii. Spre exemplu, putem spune despre un obiect autovehicul ca are ca si parte componenta un obiect motor. Pe de alta parte, putem spune ca motoarele diesel sunt un fel mai special de motoare. Din exemplul secund deriva cea mai importanta relatie ce poate exista intre doua clase de obiecte: **relatia de mostenire**. Practic, **relatia de mostenire** reprezinta **inima** programarii orientate pe obiecte.

Ierarhizarea

La fel ca notiunile de **abstractizare** si **incapsulare**, **ierarhizarea** este un concept fundamental in programarea orientata pe obiecte. Rolul **procesului de abstractizare** (cel care conduce la obtinerea unei abstractiuni) este de a identifica si separa, dintr-un punct de vedere dat, ceea ce este important de stiut despre un obiect si ceea ce nu este important. Rolul **mecanismului de incapsulare** este de a permite ascunderea a ceea ce nu este important de stiut despre un obiect. Dupa cum se poate observa, **abstractizarea** si **incapsularea** tind sa micsoreze cantitatea de informatie disponibila utilizatorului unei abstractiuni. O cantitate mai mica de informatie conduce la o intelegere mai usoara a respectivei abstractiuni. Dar ce se intampla daca exista un numar foarte mare de abstractiuni?

Intr-o astfel de situatie, des intalnita in cadrul dezvoltarii sistemelor software de mari dimensiuni, simplificarea intelegerii problemei de rezolvat se poate realiza prin ordonarea acestor abstractiuni, formandu-se astfel **ierarhii** de abstractiuni.

O **ierarhie** este o clasificare sau o ordonare a abstractiunilor.

Ierarhia pe obiecte - Relatia de agregare

Ierarhia de obiecte este o ierarhie de tip intreg "\parte". Sa consideram un obiect dintr-o astfel de ierarhie. Pe nivelul ierarhic imediat superior acestui obiect se gaseste obiectul din care el face parte. Pe nivelul ierarhic imediat inferior se gasesc obiectele ce sunt parti ale sale.

Este simplu de observat ca o astfel de ierarhie descrie relatiile de tip **part of** dintre obiecte. In termeni aferenti programarii orientate pe obiecte, o astfel de relatie se numeste **relatie de agregare**.

In secventa de cod de mai jos se poate vedea cum este transpusa o astfel de relatie in codul sursa Java.

```
class Tastatura {
    //Elemente specifice unei tastaturi
}

class Monitor {
    //Elemente specifice unui monitor
}

class Calculator {
    //Fiecare calculator trebuie sa aiba o tastatura
    Tastatura tastatura;
    //Fiecare calculator trebuie sa aiba un monitor
    Monitor monitor;

    public Calculator() {
        //O posibila solutie de instantiere
        this.monitor = new Monitor();
        this.tastatura = new Tastatura();
```

```
    }
}
```

Ierarhia de clase - Relatia de mostenire

Ierarhia de clase este o ierarhie de tip generalizare”\”” specializare. Sa consideram o clasa B care face parte dintr-o astfel de ierarhie. Pe nivelul ierarhic imediat superior se gaseste o clasa A care defineste o abstractiune mai generala decat abstractiunea definita de clasa B. Cu alte cuvinte, clasa B defineste un set de obiecte mai speciale incluse in setul de obiecte definite de clasa A. Prin urmare, putem spune ca B **este un fel de** B.

Dupa cum se poate observa, ierarhia de clase este generata de relatii de tip “is a” dintre clasele de obiecte, aceasta relatie numindu-se **relatie de mostenire**. In cadrul unei astfel de relatii, clasa A se numeste **superclasa** a clasei B, iar B se numeste **subclasa** a clasei A.

Dupa cum am spus inca de la inceput, **mostenirea** este **inima** programarii orientate pe obiecte. Este normal sa apara intrebarea: de ce? Ei bine, limbajele de programare orientate pe obiecte, pe langa faptul ca permit programatorului sa marcheze explicit relatia de mostenire dintre doua clase, mai ofera urmatoarele facilitati:

1. o **subclasa** preia (**mosteneste**) reprezentarea interna (**datele**) si comportamentul (**metodele**) de la **superclasa** pe care o mosteneste.
2. un obiect instanta a unei subclase poate fi utilizat in locul unei instante a superclasei sale;
3. legarea dinamica a apelurilor metodelor.

In cadrul acestui laborator, ne vom ocupa de primele doua aspecte enuntate, cunoscute si sub numele de **mostenire de clasa**, respectiv **mostenire de tip**.

Programarea orientata pe obiecte este o paradigma de programare in care programele sunt organizate ca si colectii de obiecte care coopereaza intre ele, fiecare obiect reprezentand instanta unei clase, fiecare clasa fiind membra unei ierarhii de clase ce sunt definite uzitand relatii de mostenire.

Relatia de mostenire in Java

Exprimarea relatiei de mostenire dintre o **subclasa** si **superclasa** ei se realizeaza in Java utilizand cuvantul cheie **extends**.

```
class nume_subclasa extends nume_superclasa {
        //continutul subclasei (membrii + metode)
}
```

Desi aceasta constructie Java exprima atat mostenirea de clasa, cat si mostenirea de tip intre cele doua clase, vom trata separat cele doua notiuni pentru a intelege mai bine distinctia dintre ele.

Mostenirea de clasa in Java

Mostenirea de clasa este o facilitate a limbajelor de programare orientate pe obiecte care permite sa definim implementarea unui obiect in termenii implementarii altui obiect. Mai exact, **subclasa** preia sau “mosteneste” reprezentarea interna (“datele”) si comportamentul (“metodele”) de la **superclasa**. Dupa cum se poate observa, acest principiu POO permite reutilizarea de cod.

Pentru a intelege mai bine conceptul de mostenire de clasa, dar si pentru o exemplificare a regulilor de vizibilitate a membrilor de clasa, se propune analizarea urmatoarei secvente de cod Java. Se poate observa ca, din perspectiva unui client, nu se face distinctie intre membrii mosteniti de o clasa si cei specifici ei.

```
class SuperClasa {
    public int super_a;
    private int super_b;
    protected int super_c;
```

```
}

class SubClasa extends SuperClasa {
    public void metoda(SuperClasa x) {
        super_a = 1; //Corect
        super_b = 2; //Eroare la compilare
        super_c = 3; //Corect

        x.super_a = 1; //Corect
        x.super_b = 2; //Eroare la compilare
        x.super_c = 3; //Corect (clase in acelasi pachet)
    }
}

class Client {
    public void metoda() {
        SuperClasa sp = new SuperClasa();
        SubClasa sb = new SubClasa();
        sp.super_a = 1; //Corect
        sp.super_b = 2; //Eroare la compilare
        sp.super_c = 3; //Corect (clase in acelasi pachet)

        sb.super_a = 1; //Corect
        sb.super_b = 2; //Eroare la compilare
        sb.super_c = 3; //Corect (clase in acelasi pachet)
    }
}
```

Cuvantul cheie super

In cazul in care exista date care au acelasi nume si in **subclasa**, dar si in **superclasa**, este posibil sa se produca anumite confuzii, datorate acestui inconvenient.

In exemplul de mai jos, ce camp de date este initializat (cel din **subclasa** sau cel din **superclasa**)? Care ar putea fi explicatia acestui raspuns?

Standardul Java prevede ca in astfel de situatii sa se acceseze campul "nr" local clasei **SubClasa**. Daca dorim sa accesam campul "mr" mostenit, vom proceda ca in exemplul de mai jos, uzitand cuvantul cheie **super**. Acesta trebuie vazut ca o referinta la partea mostenita a obiectului apelat.

```
class A {
    int nr;
}

class B extends A {
    int nr;
    public B() {
        //Obiectul din subclasa (B)
        this.nr = 1;
        //Obiectul din superclasa (A)
        super.nr = 2;
    }
}
```

Spre deosebire de alte limbaje de programare orientate-obiect, Java permite doar **mostenirea simpla**, ceea ce inseamna ca o clasa poate avea un singur parinte. Evident, o clasa poate avea oricati mostenitori (subclase), de unde rezulta ca multimea tuturor claselor definite in Java poate fi vazute ca un arbore, radacina acestuia fiind clasa **Object**.

Mostenirea de tip in Java

- **Mostenirea de tip** este o facilitate a limbajelor de programare orientate pe obiecte care permite sa utilizam o instanta a unei subclase in locul unei instante a superclasei sale.
- Pentru o mai buna intelegere, vom analiza un exemplu elocvent.

```
class Paralelogram {
    public int lungime, latime;

    public Paralelogram(int lungime, int latime) {
        this.lungime = lungime;
        this.latime = latime;
    }
}

class Patrat extends Paralelogram {
    public Patrat(int latura) {
        super(latura, latura);
    }
}

class Test {
    public static void main(String args[]) {
        Paralelogram p;
        Patrat p1 = new Patrat(10);
        p = p1;
    }
}
```

Datorita **mostenirii de tip**, acest cod este corect, deoarece este permis sa utilizam un obiect **Patrat** ca si cum ar fi o instanta de tip **Paralelogram**. Este logic de ce e permis acest lucru: subclasa mosteneste reprezentarea si comportamentul de la superclasa.

Lantul de apeluri al constructorilor

Deoarece o clasa derivata mosteneste membrii clasei de baza, atunci cand este instantiat un obiect din clasa derivata, trebuie apelat constructorul clasei de baza pentru a initializa membrii care provin din clasa de baza. Initializarea membrilor din clasa de baza trebuie realizata explicit in apelul constructorului clasei derivate. In caz contrar, pentru membrii care provin din clasa de baza se apeleaza automat constructorul implicit al clasei de baza.

In cazul in care exista numai constructori cu parametrii in clasa parinte, trebuie apelat explicit constructorul dorit din superclasa, uzitand cuvantul cheie **super**. Acest apel trebuie sa reprezinte prima instructiune din constructor (prima linie).

Pentru o intelegere mai buna a modului in care se apeleaza constructorii in cazul mostenirii, se recomanda consultarea exemplelor oferite in cadrul cursului si analizarea exemplului urmator.

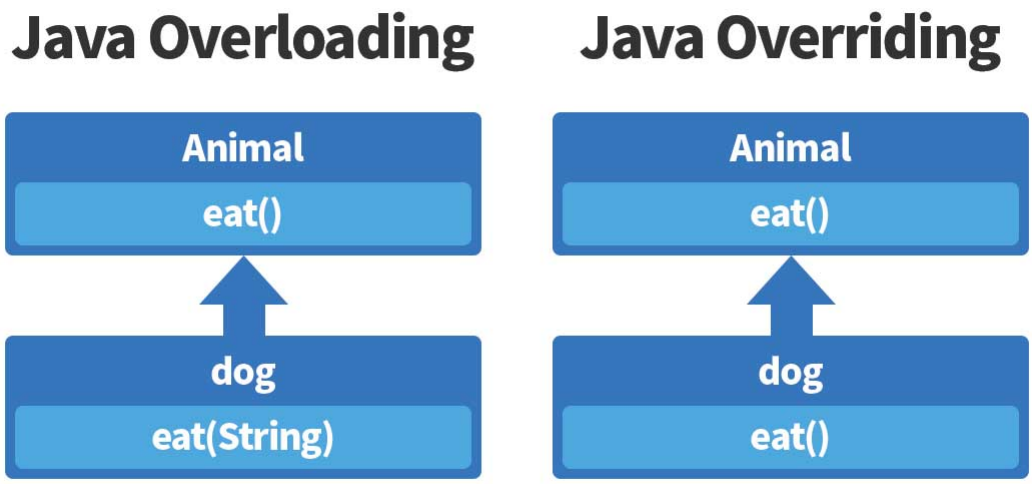
```
class D {
    public D() {
        System.out.println("Constructor D");
    }
}
class A extends D{
    public A() {
        this(10);
        System.out.println("Constructor 1 - A");
    }
    public A(int nr) {
        System.out.println("Constructor 2 - A " + nr);
    }
}
class B extends A {
    public B(int nr) {
        System.out.println("Constructor 1 - B " + nr);
    }
}
class C extends B {
    public C() {
        super(7);
        System.out.println("Constructor - C");
    }
}
class Test {
    public static void main(String args[]) {
        C obj = new C();
```

```
    }
}
```

Supraincarcarea vs Supradefinirea metodelor

Supraincarcarea si **supradefinirea** metodelor sunt doua concepte extrem de utile ale programarii orientate pe obiecte, cunoscute si sub denumirea de **polimorfism**, si se refera la:

1. **supraincarcarea (overloading)**: in cadrul unei clase pot exista metode cu acelasi nume cu conditia ca signaturile lor sa fie diferite (lista de argumente primite sa difere fie prin numarul argumentelor, fie prin tipul lor) astfel incat la apelul functiei cu acel nume sa se poata stabili, in mod unic, care dintre ele se executa.
2. **supradefinirea (overriding)**: o subclasa poate rescrie o metoda a clasei parinte prin implementarea unei metode cu acelasi nume si aceeasi signatura ca ale superclasei.



O metoda nu poate supradefini o metoda declarata **finala** in clasa parinte. De asemenea, in Java, nu este posibila supraincarcarea operatorilor.

Orice clasa care nu este abstracta trebuie obligatoriu sa supradefineasca metodele abstraste ale superclasei. In cazul in care o clasa nu supradefineste toate metodele abstracte ale parintelui, ea nu este instantiabila si va trebui declarata ca o clasa abstracta.

```
class A {
    void metoda() {
        System.out.println("A: Metoda 1");
    }
    //Supraincarcare
    void metoda(int nr) {
        System.out.println("A: Metoda 2 " + nr);
    }
}

class B extends A {
    //Supradefinire
    void metoda() {
        System.out.println("B: Metoda supradefinita");
    }
}
```